OVERTURE: Object-Oriented Tools for Solving CFD and Combustion Problems *

David L. Brown and William D. Henshaw
Scientific Computing Group CIC-19
Computing, Information, and Communications Division
Los Alamos NM, USA, 87545
{dlb,henshaw}@lanl.gov

Keywords: differential equation solvers, hydrodynamics, object-oriented frameworks, computer software.

ABSTRACT

The *Overture* Framework is an object-oriented environment for solving PDEs on serial and parallel architectures. It is a collection of C++ libraries that enables the use of finite difference and finite volume methods at a level that hides the details of the associated data structures, as well as the details of the parallel implementation. It is based on the A++/P++ array class library and is designed for solving problems on a structured grid or a collection of structured grids. In particular, it can use curvilinear grids, adaptive mesh refinement and the composite overlapping grid method to represent problems with complex moving geometry.

INTRODUCTION

The *Overture* Framework is an object-oriented C++ library for solving partial differential equations (PDEs) on serial and parallel architectures. It supports finite difference and finite volume computations on a structured grid, or on a collection of structured grids. Collections of structured grids are used, for example, in the method of composite overlapping grids, with block-structured adaptive mesh refinement (AMR) algorithms, and for patched-based domain decomposition methods. This paper concentrates on the implementation of support for the method of composite overlapping grids (Chesshire and Henshaw, 1990, Henshaw, 1996) which we use for high-resolution simulations of incompresible and low Mach number hydrodynamics flows in complex moving geometries.

A composite overlapping grid consists of a set of logically rectangular (in 2-D) or hexahedral (in 3-D) curvlinear computational grids that overlap where they meet and together are used to describe a computational region of arbitrary complexity. This method has been a successful approach for solving problems involving fluid flow in complex, often dynamically moving, geometries (Brislawn et al. 1995, Brown 1994, Dougherty and Kuan 1989, Henshaw 1994, Meakin 1997, Steger and Benek 1987).

The data structures associated with a flexible overlapping grid solver can be quite complex. Mathematically, each component grid can be described in terms of a transformation from the unit square or cube to the coordinate space of that grid. In order to complete the description of the computational geometry, the overall composite grid also requires information specifying how the component grids communicate with each other *e.g.* through interpolation formulas. It is also possible for component grids to move with respect to each other as part of a time-dependent simulation. Thus, tools are required to efficiently recompute the overlap information when the grids move. In the discrete representation of such a system, for each component grid, data such as the location of the grid points, values of the transformation derivatives and volumes of the grid cells must be stored. In addition, each grid point can have attributes associated with it, such as whether it is used for discretization of the PDE or a boundary condition, if it will have values interpolated to it from another component grid, or possibly that it is not

*This work supported by the Mathematical, Information and Computational Sciences Division of the Office of Energy Research in the U.S. Department of Energy and by the Mathematical, Computer and Information Sciences Division of the Office of Naval Research in the United States Navy.

used at all. Information on where to find interpolation stencils for the interpolation points must also be stored.

The PDEs that are to be approximated can be quite complex. The difference approximations that are used can vary from relatively simple (e.g. centered second-order finite-difference methods) to quite complex (e.g. unsplit Godunov procedures for compressible or incompressible fluid flow (Brown 1994), or fully fourth-order centered finite difference methods (Henshaw 1994). In addition, techniques such as block structured AMR may be used to locally increase computational resolution and increase overall computational efficiency. If the simulation is to run on a parallel architecture, there are correspondingly more complexities involved in writing the code. The net result of the data structures, advanced algorithms, and modern architectures is a PDE solver code that is an extremely complex system. Successfully writing, debugging, modifying and maintaining software that implements this complex system is a daunting if not impossible task using a traditional structured programming approach and procedural languages such as Fortran or C.

An alternative to the traditional structured approach is to use object-oriented design principles and object-oriented languages like C++ to write the code (Booch 1994). With object-oriented design, the task is to develop computational "objects" that represent fundamental abstractions of elements in a computational model. Where in the structured approach, the fundamental unit of code is a subroutine or function that modifies the data in some way, in the object-oriented approach the fundamental unit is an object, described by a *class* in C++. A class contains both a description of the data structures that describe the object, as well as class member functions that operate on that data. An example of an object for a composite grid application is the composite grid itself. The class describing composite grids includes a description of the data describing the grid as well as functions that operate on that data. Examples of such functions might be those that get or put the data to a database file, add an adaptive mesh refinement grid to the data structure, or return values of parameters that describe properties of the grid.

Overture is an object-oriented framework that supports applications of the type discussed above. It has been used to develop a variety of PDE solvers that use the composite overlapping grid method and support applications at Los Alamos. Among these are solvers describing incompressible, nearly incompressible and high-speed compressible fluid flow. Under development at present are solvers for internal combustion applications. The remainder of this paper discusses details of this framework and presents some computational examples.

OVERVIEW OF THE OVERTURE CLASSES

The main class categories that make up *Overture* are as follows:

- **Arrays** describe multidimensional arrays using A++/P++. A++ provides the serial array objects, and P++ provides the distribution and interpretation of communication required for their data parallel execution. (Quinlan 1995)
- **Mappings** define transformations such as curves, surfaces, areas, and volumes. These are used to represent the geometry of the computational domain.
- **Grids** define a discrete representation of a mapping or mappings. These include single grids, and collections of grids; in particular composite overlapping grids.
- **Grid functions** storage of solution values, such as density, velocity, pressure, defined at each point on the grid(s).
- **Operators** provide discrete representations of differential operators and boundary conditions
- **Plotting** provides high-level plotting interface based on OpenGL.
- **Adaptive Mesh Refinement:** The AMR++ library provides support for block-structured adaptive mesh refinement.
- **Load Balancing:** The MLB load balancing library is presented in (Quinlan and Berndt 1997)

Solvers for partial differential equations are written using the above classes.

THE A++ AND P++ ARRAY CLASSES

A++ and P++ (Quinlan 1995) are array class libraries for performing array operations in C++ in serial and parallel environments, respectively. P++ is the principle mechanism by which the *Overture* Framework operates in parallel, there is little code in *Overture* outside of P++ which is specific to parallel execution.

A++ is a *serial* array class library similar to FORTRAN 90 in syntax, but not requiring any modification to the C++ compiler or language. A++ provides an object-oriented array abstraction specifically well suited to large scale numerical computation. It provides efficient use of multidimensional array objects which serves to both simplify the development of numerical software and provide a basis for the development of parallel array abstractions. P++ is the *parallel* array class library and shares an identical interface to A++, providing a simple and elegant mechanism that allows serial code to be reused in the parallel environment.

Here is a simple example code segment that solves Poisson's equation with the Jacobi method in either a serial or parallel environment using the A++/P++ classes. Notice how the Jacobi iteration for the entire array can be written in one statement.

```
  // Solve u_xx + u_yy = f by a Jacobi Iteration
  // ... define a range of indices: 0,1,2,...,n
Range R(0,n)
  // ... declare two two-dimensional arrays
floatArray u(R,R), f(R,R)
  // ... initialize arrays and parameters
f = 1.; u = 0.; h = 1./n;
 // ... define ranges for the interior
Range I(1,n-1), J(1,n-1);

for (int iteration=0; iteration<100; iteration++)
  // ... data parallel
  u(I,J) =
    .25*(u(I+1,J)+u(I-1,J)+u(I,J+1)
                 +u(I,J-1)-f(I,J)*(h*h));
```

In this example, "Range" objects are first constructed using base, bound, and optional stride information. These are then used to build the array objects and later to specify the indexing in the final array statement. In a parallel environment, the "`for`" loop is executed on all processors and the statement representing the Jacobi relaxation step executes using an SPMD simulation of data parallel execution. Communication requirements are interpreted at runtime as required to permit the dynamic redistribution of data (required for AMR applications).

MAPPINGS AND GRIDS

The geometry of the computational domain is defined by a set of mappings, one mapping for each grid. Mappings have been designed so that an object can be easily moved by composing it with a transformation such as a translation, rotation or scaling. In general, a mapping defines a transformation from R^n to R^m. In particular, mappings can define lines, curves, surfaces, volumes, rotations, coordinate stretchings , etc. The base class `Mapping` contains the data and functions that apply to all mappings. Specific types of mappings are derived from this base class. Mappings contain a variety of information and functions that can be useful for grid generators and solvers. For example, mappings contain information about their domain space, range space, boundary conditions and singularities. Mappings are easily composed, allowing coordinate stretching, rotations, translations, bodies of revolution, etc. The inverse of a mapping is always defined, either analytically or by discrete approximation.

Grids define a discrete representation of a mapping. There are several main grid classes. The `MappedGrid` class defines a grid for a single mapping that contains, among other things, a mapping and a mask array for cut-out regions. The `GridCollection` class defines a collection of `MappedGrid`'s. The `CompositeGrid` class defines a valid overlapping grid, which is essentially a `GridCollection` plus interpolation information. Grids contain many geometry arrays such as grid points, Jacobians, normal vectors, face areas and cell volumes.

GRID FUNCTIONS

Grid functions represent solution values at each point on a grid or grid-collection. There is a grid function class (of `float`'s, `int`'s or `double`'s) corresponding to each type of grid. So, for example, a `MappedGridFunction` lives on a `MappedGrid` and a `CompositeGridFunction` lives on a `CompositeGrid`. Grid functions are defined with up to three coordinate indices (i.e. up to three space dimensions) and up to five component indices (i.e. they can be scalars, vectors, matrices, 3-tensors,...). Since they are derived from A++ arrays, all of the array operations are defined. In the following example, a grid function is made and assigned values at all points on the grid.

```
  // ... create a mapping
SphereMapping sphere;
  // ... the sphere  mapping has been used to define a grid
MappedGrid mg (sphere);
  // ... this function computes all the geometry arrays
mg.update();
  // ...other grid function centerings can be
  // ...specified through this class
GridFunctionParameters defaultCentering;

  // ... create a grid fn with  default centering and
  // ... 2 components defined at all grid points
floatMappedGridFunction  u(mg,defaultCentering,2);
Index I1,I2,I3;
  // ... get Index'es for all grid points
```

```
getIndex(mg.dimension,I1,I2,I3);

  // ... set x-component to sin(x)*cos(y)
const int xComp = 0, yComp = 1;
u(I1,I2,I3,xComp) = sin(mg.vertex(I1,I2,I3,xComp))
                    *cos(mg.vertex(I1,I2,I3,yComp));
```

Notice that when the `floatMappedGridFunction` is declared, the number of grid points does not have to be specified since this information is contained in the `MappedGrid`.

OPERATORS

Operators define discrete approximations to differential operators and boundary conditions for grid functions. Many different types of approximations can be used. For example, the class `MappedGridOperators` defines finite-difference style operators, while the class `MappedGridFiniteVolumeOperators` defines finite-volume style operators. Operator classes for compressible and incompressible flow Godunov methods have also been implemented. The `Projection` class computes the divergence-free part of a velocity function and is used in some of our incompressible flow codes. Here is an example using one of the operator classes:

```
...
MappedGrid mg(sphere);
  // ...define operators for a MappedGrid
MappedGridFiniteVolumeOperators op(mg);
floatMappedGridFunction u(mg), v(mg);
  // ...associate operators with grid fn.
u.setOperators (op);
  // ...assign u some values
u = ...
  // ...compute gradient of u
v = u.grad();
  // ...compute Laplacian(u)
v = u.laplacian();
  // ...compute sparse matrix for the discrete Laplacian
v = op.laplacianCoefficients();
```

The result of the statement `u.grad()` is a grid function containing the gradient of `u`. An equivalent statement is `op.grad(u)`. The matrix for the discrete Laplacian holds the stencil at each grid point for the Laplacian, and so is a grid function itself. This grid function can be passed to a sparse solver, for example

BOUNDARY CONDITIONS

The programming model for boundary conditions is to use ghost points (instead of one-sided difference approximations). A library of elementary boundary conditions such as Dirichlet, Neumann, extrapolation, etc has been defined. Solvers define more complicated boundary conditions in terms of these elementary ones. The interface is quite simple. For example, the following statements set all components of the velocity grid function `v` to zero on all boundaries of type "`wall`".

```
floatCompositeGridFunction v;
v.applyBoundaryCondition
  (allVelocityComponents, dirichlet, wall, ZERO);
```

A COMPLETE CODE

This example demonstrates the power of the *Overture* Framework by showing a working code that solves the incompressible Navier-Stokes equations in any number of space dimensions on an overlapping grid. It is based on a cell-centered Projection method with a two-stage Runge-Kutta time integrator. A routine to initialize the velocity, `initializeVelocity`, and to initialize the Projection boundary conditions, `setProjectionBoundaryConditions`, must also be supplied to complete the code. `PlotStuff` is the graphics package associated with *Overture*.

```
main ()
{
    //...create composite grid
  CompositeGrid cg;
    //...read in from database (HDF) file
  getFromADataBase (cg, "grid.hdf");
  cg.update ();
    // ... initialize interpolant
  Interpolant interp (cg);
    // ... initialize plotting
  PlotStuff ps (TRUE);
    // ... plot the grid
  ps.plot (cg);
    // ... velocities stored in q,qMid
  int nComp = 2;
  floatCompositeGridFunction q    (cg, defaultCentering, nComp);
  floatCompositeGridFunction qMid (cg, defaultCentering, nComp);
  initializeVelocity (vortexInBox, q, cg);
  CompositeGridFiniteVolumeOperators op (cg);
  q.setOperators (op);
  qMid.setOperators (op);
    // ... initialize Projection operator
  Projection projection (cg);
  setProjectionBoundaryConditions (projection);

    // ... solve Incompressible Navier-Stokes equations
  float t=0., dt=.0005, viscosity=.05;
  int numberOfSteps=100, frequencyOfOutput = 10;
  for (int step=0; step < numberOfSteps; step++)
  {     //... forward Euler prediction
    qMid = q + 0.5*dt*(-q.convectiveDerivative()
                  + viscosity*q.laplacian());
    applyVelocityBoundaryConditions (qMid);
        // ... velocity projection
    qMid = projection.project (qMid);
      // ... midpoint rule prediction
    q = q + dt*(-qMid.convectiveDerivative()
              + viscosity*qMid.laplacian() );
    applyVelocityBoundaryConditions (q);
      // ... correct again with projection
    q = projection.project (q);
      // ... plot every so many timesteps
    if (step % frequencyOfOutput == 0)
      ps.streamLines (q);
  }
}
```

NUMERICAL COMPUTATION OF FLOW THROUGH A VALVE

Figure 1 shows a computation of the incompressible Navier-Stokes equations for flow through a three-dimensional valve that is nearly closed. The overlapping grid is shown in Figure 2. The computation was done using an incompressible Navier-Stokes solver written using the *Overture* Framework. The fluid is moving from the bottom to the top. The pressure is plotted on the surface of the valve. The flow speed is plotted on two planes that intersect the computational region. The computational region is cylindrically symmetric but no symmetry is used in the computation.

SOFTWARE AVAILABILITY

The *Overture* Framework and documentation is available for public distribution at the Web site `http://www.c3.lanl.gov/cic19/teams/napc/`. A++/P++ dates back to its first version in 1990 and has been publicly distributed since 1994; the current version was released in 1996 (Quinlan 1996). The *Overture* libraries have been under development since 1994, and have been available to the public since 1996 (Brislawn et al. 1996). The AMR++ classes in *Overture* are still under development and are expected to be released during 1998.

REFERENCES

Booch, G. 1994, *Object-oriented analysis and design with applications*, Addison-Wesley, 2nd ed.

Brislawn, K. D., Brown, D. L., Chesshire, G. S., and Quinlan, D. J. 1996, *Overture code*, Los Alamos National Laboratory Computer Code LA-CC-96-04, Los Alamos, NM.

Brown, D. L. 1994, "An unsplit Godunov method for systems of conservation laws on curvilinear overlapping grids", *Math. Comput. Modelling*, 20: 29–48.

Chesshire, G. and Henshaw, W. D. 1990, " Composite overlapping meshes for the solution of partial differential equations", *J. Comp. Phys.*, 90: 1–64.

Henshaw, W. D., 1994, " A fourth-order accurate method for the incompressible Navier-Stokes equations on overlapping grids", *J. Comp. Phys.*, 113: 13–25.

Henshaw, W. D., 1996, *Ogen: an overlapping grid generator for Overture*, LANL unclassified report 96-3466, Los Alamos National Laboratory, Los Alamos, NM.

Meakin, R. 1997, *On adaptive refinement and overset structured grids*, AIAA-97-1858-CP, 13th AIAA Computational Fluid Dynamics Conf., Snowmass, CO: 236-249 (June).

Quinlan, D., 1993, *Adaptive Mesh Refinement for Distributed Parallel Processors*, PhD thesis, University of Colorado, Denver, CO (June).

Quinlan, D., 1995, *A++/P++ manual*, LANL Unclassified Report 95-3273, Los Alamos National Laboratory, Los Alamos, NM.

Quinlan, D., 1996, *A++/P++ class libraries*, LANL Computer Code LA-CC-96-01, Los Alamos National Laboratory, Los Alamos, NM.

Quinlan, D. and Berndt, M. 1997, *MLB: Multi-level load balancing*, Proceedings of the SIAM 1997 Conference on Parallel Processing, Society for Industrial and Applied Mathematics, Philadelphia, PA.

Steger, J. L. and Benek, J. A. 1987, " On the use of composite grid schemes in computational aerodynamics",*Computer Methods in Applied Mechanics and Engineering*, 64: 301–320.

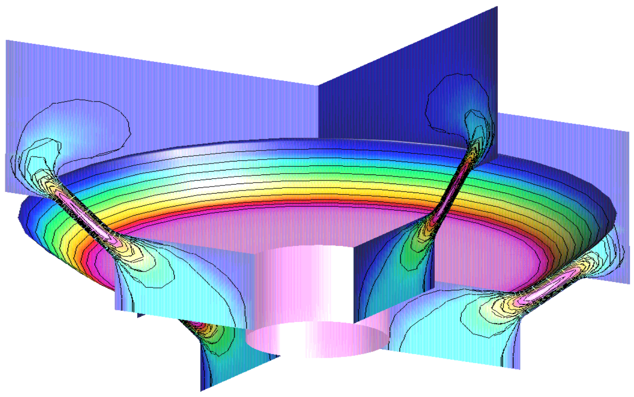

Figure 1: Flow through a three-dimensional valve.

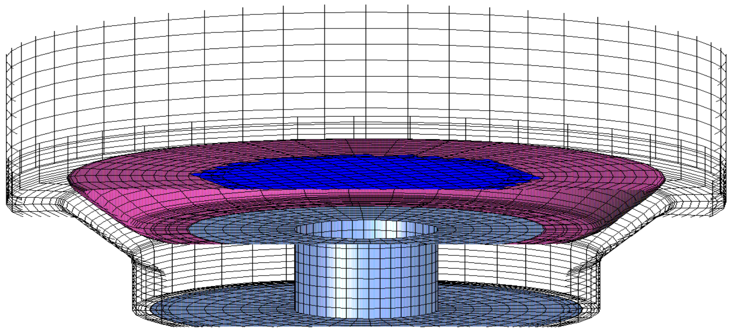

Figure 2: Grid for a three-dimensional valve.